C ++ CODING STANDARDS

1. INTRODUCTION

We can judge the quality of source code along several axes: satisfaction of functional and performance requirements, correctness (that is, the absence of defects or bugs), and ease of maintenance. When code is easier to understand, it is easier to maintain.

Coding conventions help us write code that is understandable. When code is understandable, it can be reviewed, maintained, and extended more easily. Coding conventions are useful when many programmers are working on a project, when a programmer other than the original author will take over a body of code for maintenance, and for communicating effectively during code reviews. Coding guidelines can also help reduce certain kinds of coding errors.

Because this document is short, rationale and many nuances are not covered. Refer to the following references for additional guidance.

Cargill: Tom Cargill, C++ Programming Style, Addison-Wesley, 1992.

Ellemtel: Programming in C++, Rules and Recommendations, Ellemtel Telecommunications Systems Laboratories, 1992. Herein, references are made to Rules, Recommendations (Rec.), and Portability Recommendations (Prt. Rec.).

FSL Conventions: Forecast System Laboratory C/C++ Software Implementation Conventions, Draft, Robert Prentice et al, 9/28/92.

Koenig: Andrew Koenig, C Traps and Pitfalls, Addison-Wesley, 1988.

McConnell: Steve McConnell, Code Complete, Microsoft Press, 1993, Chapter 9, The Power of Data Names.

Meyers: Scott Meyers, Effective C++, 50 Specific Ways to Improve Your Programs and Designs, Addison-Wesley, 1992.

Where the references differ, this document prevails.

Remember, the overriding consideration is communication. As software authors we have a dual audience: the compiler and other people.

2. STANDARDS

Adhere to relevant standards.

    a. For C language programs, use ANSI C. (This implies that function
       prototypes are required.) Use the -Aa option in both the HP C and C++
       compilers for ANSI C.
    b. For C++, the bible is the ARM, that is, the Annotated Reference Manual
       by Ellis and Stroustrup. We will attempt to maintain consistency with
       the progress of the C++ WG21 evolving draft standard.
    c. Use standard libraries. For example, learn the functionality of the C
       Standard Library and use those functions, not your own. Include
       (#include) the header files that come with the libraries you use.
       Don't retype the function prototypes.
    d. Comply with POSIX standards where reasonable.
    e. Avoid vendor-specific extensions.

3. NAMES

Choice of names is the most important single style element for
understandability. Names should be meaningful. Choose a verb for a function
name that reflects the action taken by the function. (An exception is
listed as a rule below.) Choose a noun for a variable name. Choose a file name
to reflect the content of the file.

McConnell has a good description of how to choose variable names. He states:

"The most important consideration in naming a variable is that the name fully
and accurately describe the entity the variable represents. An effective
technique for coming up with a good name is to state in words what the
variable represents. Often that statement itself is the best variable name..."
(p. 186)

"When you find yourself `figuring out' a section of code, consider renaming
the variables. It's OK to figure out murder mysteries, but you shouldn't need
to figure out code. You should be able to read it." (p. 193)

Choose names to reflect elements of the problem domain rather than the
computational solution. For instance, SAOreport is a better name than
inputRecord for a variable that is an input record containing an
SAO report.

The length of a name should reflect its scope. A local variable in a small
function has very limited scope. Here, max might be a suitable name although
maxTemp is probably better. A global variable has wide scope. Globally, we
might need a name like maxTempForMostRecentSAO.

As a practical matter, limit the names of object files to 14 characters, the
names of C and C++ identifiers to 31 characters, and do not begin a name with
an underscore (except as noted in point i below) or a double underscore.

See McConnell (Chap. 9, esp. Sec. 9.2) for further discussion of the following
items.

    a. When using opposites, be precise. For example use first/last, not
       first/end.

2

b. Use meaningful loop index names (as shown in the example below) for long (>15 lines) or nested loops. Loop index names such as i are acceptable for short loops.

```
for (windowIndex = 0; windowIndex <= windowCount; windowIndex++)
    { /* ... */ }
```

c. "Think of a better name than flag for status variables. CharacterType = CONTROL_CHARACTER is more meaningful than StatusFlag = 0x80."
d. Think twice about short variable names, for example, temp. Is it temperature? Is it temporary? Temporary what? If it's not clear from context, use a better name.
e. "Give boolean variables names that imply True or False." For instance, success is initialized to False and set to True when the related operation has succeeded.

Now for some boring but helpful rules:

f. Use UPPERCASE_IDENTIFIERS for #defined constant and macro identifiers, for constants, for the values of an enumerated type (enumerators), and for global variables (that is, variables that are visible in more than one linkage unit).
g. Use UpperMixedCase (where the first character is upper case) for user defined types (including enumerations, classes, structs, and unions) and for names of static variables with file scope.
h. Use lowerMixedCase (where the first character is lower case) for local variable names, function names, formal parameter names, and members of C language structs and unions.
i. Use_lowerMixedCase (where the first character is an underscore and the next is lower case) for member variables of C++ classes. This helps the reader to distinguish a member variable in a member function.
j. When using typedef, reflect the style appropriate to the underlying type. For example, use lowerMixedCase for a typedef-name for a function or a built-in type and UpperMixedCase for a typedef-name for a class, struct, or union. If the purpose of the typedef is to transparently switch between underlying types, choose any appropriate naming style.
k. Exception for C++ access function names: Function names are generally verbs. However, when a member function returns the value of a member variable, the name of the variable sans the leading underscore is the preferred name for the function. For example, if class Parcel has a member named _temperature, then Parcel::temperature() is preferred over Parcel::getTemperature().
l. Use only generally accepted abbreviations.
m. Sometimes it is appropriate to use an uppercase abbreviation in a mixed case name. If so, you may use an underscore as a separator between words. SAO_Report is preferred although SAOreport and saoReport are both acceptable.
n. All identifiers that have global scope in a function library should have a lowercase, two character prefix separated from the identifier by an underscore. For example, gr_newSegment() could be the name of a function in the graphics library. Because the name space is encapsulated, names in a class library need not have a prefix.
o. Use UpperMixedCase for the name of a file that declares or defines a C++ class. Otherwise, use lowerMixedCase for file names. Suffixes for source code file names are .c for C language definition files, .h for C language include files, .C for C++ definition files, and .H for C++ include files.

    p. Write numbers in decimal, unless there is a good reason to use another base. Clearly document any numbers based upon underlying hardware or software services.

    q. Names of formal parameters should appear in both the function prototype and the function definition. If a function has no formal parameters, use the void keyword instead of an empty parameter list (to compensate for the inconsistency between ANSI C and C++).

    Note that many of the above rules are based on, but not identical to, the Identifier Format Conventions in the FSL Conventions.

4. SAFE CODING

Safe coding conventions help to catch errors.

    a. Clearly document any assumptions made by functions in the comment block that precedes the function body.

    b. Be sure that every variable is initialized before use.

    c. Avoid type-casts whenever possible. When casting is unavoidable, an explicit cast is preferred over an implicit (compiler provided) one.

    d. Select restrictive argument types. This primarily includes the keywords unsigned and const, and using enumerations in preference to simple integers.

    e. Be sure that all functions have an explicit return type, even if that type is void.

    f. "A ... function must never return a reference or a pointer to a local variable." (Ellemtel Rule 34.) This rule applies to all C functions and to public C++ functions.

    g. "Do not write code which is dependent on the lifetime of a temporary object." (Ellemtel Prt. Rec. 16.)

    h. "Do not allocate memory and expect that someone else will deallocate it later." (Ellemtel Rec. 58.)

    i. "Always assign a new value to a pointer that points to deallocated memory." (Ellemtel Rec. 59.)

    j. "Always use inclusive lower limits and exclusive upper limits." (Ellemtel Rec. 52.) Also see Koenig Sec. 3.6 and Formatting point f in Section 8.

    k. If a C function has no parameters, use the keyword void to ensure consistency with C++. For example, int fnThatTakesNoParams(void);

    l. Use an explicit status parameter for a function that returns a status. Do not overload data-passing parameters or the function return value. That is, do not use a special, reserved data value to indicate a return status.

    m. Other error handling conventions are yet to be determined.

5. STRUCTURE

    a. Declare identifiers in a reasonably small scope. See Ellemtel Rec. 57.

    b. Strive for low coupling between cohesive modules. "Coupling measures the amount of interaction and dependency between components... Lower coupling, that is, fewer and simpler interactions, characterizes software that is easier to understand and maintain." (Cargill, p. 82.) "Each class [or module] should serve a single, coherent purpose." (Cargill, p. 174.)

    c. Write cohesive functions. A function should do only one thing and be fairly short. Functions over 100 lines should be relatively rare, but there are cases where this is unavoidable (such as a switch statement

with numerous branches, each containing only a line or two).

d. "Avoid functions with many arguments." (Ellemtel Rec. 41.) Three or fewer arguments is good; five or more is questionable.

e. Use symbolic names for literal constants that appear in procedural code. For example, use PI instead of 3.14159, MAX_SAO_COUNT instead of 300, and ACK instead of `\6'. Exceptions include zero, very common strings (such as "rb" in an fopen call), and NULL in C++ (see point s in Section 7). Symbolic identifiers document the code and often make it trivial to make changes. Use appropriately scoped const variables or enumerations. Use of #define is discouraged.

f. Performance optimization takes time and often results in violation of encapsulation boundaries. "Optimize code only if you know that you have a performance problem. Think twice before you begin." (Ellemtel Rec. 1.) "Don't guess. Use an execution profiler to isolate performance problems." (Cargill, Chap. 7.)

6. PORTABILITY

a. "Never specify relative UNIX names in #include directives." (Ellemtel Rule 10.) Use the #include "filename" for user-prepared include files and #include <filename for include files from libraries. (Ellemtel Recs. 10 and 11.)

b. "Avoid the direct use of [built-in] data types in declarations." (Ellemtel Prt. Rec. 1.)

c. "Do not assume that an int and a long have the same size." (Ellemtel Prt. Rec. 2.)

d. "Do not assume that an int is 32 bits long (it may be only 16)." (Ellemtel Prt. Rec. 3.)

e. "Do not assume that a char is signed or unsigned." (Ellemtel Prt. Rec. 4.)

f. "Place machine-dependent code in a special file so that it may be easily located when porting code from one machine to another." (Ellemtel Rec. 5.) This may also apply to software-package-dependent code.

7. C++ SPECIFIC CONSIDERATIONS

Most of the C conventions carry over to C++. Additional conventions include:

a. "A class should define a set of objects... Concentrate on common abstractions in a base class... A public derived class should be a specialization of its base." (Cargill, Chap 1.)

b. "Define a class interface consistently - avoid surprises." (Cargill, Chap. 2.)

c. Define constructors such that any newly constructed object is in a well defined state. See Cargill, Chap. 2.

d. "Avoid inheritance for parts-of relations." (Ellemtel Rec. 37.) "Recognize inheritance for implementation; use a private base class or (preferably) a member object [as an alternative]." (Cargill, Chap 3.)

e. Pass an argument (of a heavyweight user-defined type) via a pointer if the function will modify the argument value; otherwise, use a constant reference (const &). This reduces parameter passing overhead and helps to distinguish parameters used for input from those used for output. See Ellemtel Recs. 42 and 43.

f. For efficiency, "minimize the number of temporary objects that are created as return values from functions or as arguments to functions." (Ellemtel Rec. 46.)

g. Declare copy constructor and assignment operator member functions for

5

any class that contains any data which cannot simply be bitcopied. See Meyers, Item 11.

h. "Use operator overloading sparingly and in a uniform manner." (Ellemtel Recs. 35, 36 and 44.) Paraphrasing Cargill (Chap. 5), the meaning of overloaded operators should be natural and consistent; the set of overloaded operators should be complete.

i. Declare operator= to take const T& as the right-hand-side parameter and to return T&. See Meyers, Item 15. This differs from the Ellemtel solution.

j. "When defining operator=, remember x = x." (Cargill, Chaps. 2 and 5.) Also see Ellemtel Rule 28.

k. "A public member function must never return a non-const reference or pointer to member data." (Ellemtel Rule 29.)

l. Use inline functions rather than #define to obtain more efficient code. See Ellemtel Rule 35. However, carefully consider the impact of making a function inline. The performance consequences can be surprising.

m. "Usually, the destructor in a public base class should be virtual." (Cargill, Chap. 4.)

n. If a member function has been declared virtual in a base class, include the virtual keyword in any declaration of that function in a derived class.

o. Declare member variables as private or protected. Use access member functions to access to non-public values. See point k in Section 3.

p. "If possible, always use initialization instead of assignment." (Ellemtel Rule 41.) But, do not write code that depends upon the order of initialization. See Meyers, Item 13.

q. "Do not assume that static objects are initialized in any special order." (Ellemtel Prt. Rec. 15.)

r. Within a class declaration, declare public elements defined first, protected elements next, and private elements last. Within each visibility category, declare any constructors first and any destructor next.

s. Do not use the symbol NULL. Use 0 instead. See Ellemtel Rule 42.

t. "Do not use malloc, realloc or free." Use new and delete instead. (Ellemtel Rule 50.)

u. "Always provide empty brackets ("[]") for delete when deallocating arrays." (Ellemtel Rule 51.)

v. Avoid multiple inheritance. See Cargill, Chap. 9 and Meyers, Item 43.

w. Global functions should be fully scoped, that is, prepend the :: operator.

x. In a .H file, avoid including a file that defines a class when a forward declaration would be sufficient (that is, when the class is accessed only via pointers and references). See Ellemtel Rule 9.

y. "No member functions are to be defined within the class definition." (Ellemtel Rule 21.)

z. Stream IO is preferred to C standard IO. At any rate, avoid mixing the two.

aa. Global new() will be defined in the foundation library and will handle out-of-memory errors. Include the appropriate file and don't bother to check the return value from new().

8. FORMAT

Imposing constraints on the format of syntactic elements makes the code easier to read because it has a consistent appearance.

a. Keep source line length to 79 characters or less.
b. Write each statement on a separate line.
c. Use 4 spaces to indent code. Avoid tabs – they are interpreted differently by different editors and printers.
d. The preferred indentation for a continuation line is 2 additional spaces.
e. "Use parentheses to clarify the order of evaluation for operators in expressions." (Ellemtel Rec. 56.)
f. The following preferred brace placement and indentation conventions are a slight relaxation of the FSL Conventions.

```
for (i = 0; i < MaxCount; i++)
    {
    /* body of loop */
    }

typedef struct {
    /* body of struct */
    } name;

class example {
    public:
        example();
        ~example();
        int value() const;
        void setValue(int val);
    private:
        int _value;
    };
```

g. If a loop does all of its work in the control statement, acknowledge the null statement by placing the closing semicolon on a separate line. Alternately, a set of empty braces may be used.

```
while ((c = getchar()) != NULL)
    ;
```

h. Indent switch statements as follows:

```
switch (ch)
    {
    case `\n':
    case `\r':
        /* code */
        break;
    case `\0':
        /* code */
        break;
    default:
        /* code */
        break;
    }
```

Multiple case statements may share a common clause provided all of the clause is shared. Include a break statement in each clause. Always include a default clause.
i. Goto statements are discouraged.

9. COMMENTS

Comments should add value. Be clear and concise. Don't bother to repeat the obvious.

    a. Write comments in grammatically correct English.
    b. Provide block comments for all functions and for blocks of code that merit explanation. Make block comments stand out by separating them from code with an extra comment line or a blank line. For example:

```
/*
 *  This is a K&R block comment. Many variants exist,
 *  but there should be white space separating comment
 *  text from code.
 */
```

    c. Embedded comments, for example to clarify control flow, do not require emphasis as do block comments.
    d. In C++, use // to begin a comment.

```
// This is a C++ embedded comment. It is not separated
// from code by white space.
```

    e. Comment individual lines of code as necessary. For example, in a Finite State Machine class:

```
void reset();        // move to start state
void advance(char);  // advance one state transition
```

    f. Avoid comments to indicate what closing braces close unless the comment contributes significantly to the understandability of the code.
    g. Header comments are discussed in a separate document entitled FX-ALPHA C and C++ Header Templates.

10. ADDENDUM 1 (12/27/94)

These items augment the original 4/28/94 version of this document.

    Variable Naming

    a. <units> may be appended to the end of variable names that represent meteoro-logical values. Examples are: range_km, maxTemp_C, and height_ftMSL.
    b. _UpperMixedCase may be used for class static member variables. See also item 3.f.

    Function Parameters

    c. Give formal parameters meaningful names. Use the same formal parameter names in the function prototype and the definition. See also item 3.q.
    d. If a function returns only one value, use the function return value.
    e. If a function returns a success-or-failure status, success will be represented by the bool value true and failure by the bool value false.
    f. If a function will return a value via one or more function parameters, those parameters should appear after all input parameters for which there are not default values.

g. In C++, use pointers for those function parameters that return a value
   and use pass-by-value or const references for parameters that do not
   return a value.
h. See also Ellemtel Chap. 9, Functions.

   Directory Naming

i. The main criteria for choosing a directory name is that, in the context
   of its location in the directory hierarchy, it conveys to the reader a
   clear understanding of its contents. This is more difficult with the
   artificially flat directory structure imposed by RECOURSE.
j. Directory names are lowerMixedCase, that is, they begin with a
   lower-case character and then used mixed case as appropriate.

   File Naming (See item 3.o.)

k. Use the .H suffix for an include file that can be used only by C++
   includers. Use the .h suffix for an include file that can by used by C
   or C++ includers. If you need the C++ compiler to compile a file,
   then that file is considered to be a C++ implementation. Otherwise it is
   a C implementation. Use the .C suffix for a C++ implementation. Use the
   .c suffix for a C implementation.
l. There is generally only one class per .H/.C pair. The name of a .H/.C
   pair is exactly the class name which must begin with an uppercase
   alphabetic character.
m. Use lowerMixedCase for the names of C language files. (As for variable
   names, acronyms may be in uppercase.)
n. While function names are generally verbs, C language file names are
   generally nouns. For example, use the file name SAO_Decoder for a file
   in which the primary function is named decodeSAO().

   Libraries

o. Provide at most one library file (archive) per source-code directory.
p. Library files are named as follows: libxxx.a where xxx is identical to
   the directory name associated with the library and always begins with a
   lowercase alphabetic character. (This convention is currently violated
   in Backbone-1.)
q. When the string "Lib" appears in a directory name, it may be omitted
   from the library name. For example, the library for the directory uiLib
   may be named libuiLib.a or uilib.a.

   Environment Variable Naming

r. Environment variables that expand to path names should expand to an
   absolute rather than a relative path name. For example,
   /data/fx-2/netcdf is a good name; ../data a bad one.
s. Assume that environment variables expanding to path names omit the
   trailing slash. For example, assume ~fxa/data; not ~fxa/data/.

11. ADDENDUM 2 (1/19/95)

These items augment the revised 12/27/94 version of this document.

   Global Variables

a. Do not use global variables (that is, those that are visible in more than one linkage unit) without good reason; their use is strongly discouraged. Global variables are difficult to track, make it difficult to change or correct code without changing all clients of the code, make debugging more difficult, and are tough for compilers to optimize. They are generally considered bad style, especially for large multi-developer projects like ours. See also item 5a.

b. If you have a good reason to use a global variable, adhere to the naming conventions that are described in the introduction to Section 3. The requirement to make global variable names UPPERCASE has been struck from item 3.f (as of the date of this addendum). Instead, use a prefix as described in item 3.n for global functions.

File-Scoped Variables

c. Use file-scoped variables (that is, variables that are visible in a single linkage unit) with care. They are useful when a variable is to be shared by many functions within a module (linkage unit). They are dangerous because, as for global variables, it is easy to assign a value but difficult to determine where that assignment took place when debugging.

d. As for global variables, use a meaningful prefix for file-scoped variables to indicate the scope. For example, use prefix m for module.